SOFAS: A lightweight architecture for software analysis as a service

Ghezzi, Giacomo ; Gall, Harald C

Abstract: Access to data stored in software repositories by systems such as version control, bug and issue tracking, or mailing lists is essential for assessing the quality of a software system. A myriad of analyses exploiting that data have been proposed throughout the years: source code analysis, code duplication analysis, co-change analysis, bug prediction, or detection of bug fixing patterns. However, easy and straight forward synergies between these analyses rarely exist. To tackle this problem we have developed SOFAS, a distributed and collaborative software analysis platform to enable a seamless interoperation of such analyses. In particular, software analyses are offered as RESTful web services that can be accessed and composed over the Internet. SOFAS services are accessible through a software analysis catalog where any project stakeholder can, depending on the needs or interests, pick specific analyses, combine them, let them run remotely and then fetch the final results. That way, software developers, testers, architects, or quality assurance experts are given access to quality analysis services. They are shielded from many peculiarities of tool installations and configurations, but SOFAS offers them sophisticated and easy-to-use analyses. This paper describes in detail our SOFAS architecture, its considerations and implementation aspects, and the current set of implemented and offered RESTful analysis services.

DOI: https://doi.org/10.1109/WICSA.2011.21

SOFAS: A Lightweight Architecture for Software Analysis as a Service

Giacomo Ghezzi and Harald C. Gall
s.e.a.l. – software evolution and architecture lab
Department of Informatics
University of Zurich, Switzerland
{*ghezzi, gall*}*@ifi.uzh.ch*

***Abstract*—Access to data stored in software repositories by systems such as version control, bug and issue tracking, or mailing lists is essential for assessing the quality of a software system. A myriad of analyses exploiting that data have been proposed throughout the years: source code analysis, code duplication analysis, co-change analysis, bug prediction, or detection of bug fixing patterns. However, easy and straight forward synergies between these analyses rarely exist. To tackle this problem we have developed *SOFAS*, a distributed and collaborative software analysis platform to enable a seamless interoperation of such analyses. In particular, software analyses are offered as RESTful web services that can be accessed and composed over the Internet. *SOFAS* services are accessible through a software analysis catalog where any project stakeholder can, depending on the needs or interests, pick specific analyses, combine them, let them run remotely and then fetch the final results. That way, software developers, testers, architects, or quality assurance experts are given access to quality analysis services. They are shielded from many peculiarities of tool installations and configurations, but *SOFAS* offers them sophisticated and easy-to-use analyses. This paper describes in detail our SOFAS architecture, its considerations and implementation aspects, and the current set of implemented and offered RESTful analysis services.**

I. Introduction

Data about software development has been primarily used for supporting activities such as retrieving previous versions of the source code or examining the status of a change request or a defect. However, studies have highlighted the value of collecting and analyzing this diverse source of data. Researchers have come up with several analyses techniques: various static and dynamic code analyses, code clone detection, co-change analysis, bug prediction, or detection of bug fixing patterns. Yet, each of these studies has built its own methodologies and tools to extract, organize and utilize such data to perform their research. As a consequence, easy and straight forward synergies between these analyses/tools rarely exist due to their stand-alone nature, their platform dependence, their different input and output formats, and the variety of systems to analyze. Therefore, despite this richness, we still lack ways to effectively and systematically share and integrate data coming from different analyses and providers.

To tackle these problems we introduced a lightweight and flexible platform called *SOFAS* (SOFtware Analysis Services) [12]. It offers *distributed and collaborative software analysis services to allow for lightweight interoperability of analysis tools across platform, geographical and organizational boundaries*.

Tools are categorized in our software analysis taxonomy; they have to adhere to specific meta-models and ontologies and offer a common service interface that enables their composite use over the Internet. These distributed analysis services are accessible through an incrementally augmented software analysis catalog. The main purpose of *SOFAS* is to offer a single entry point to these software analyses. A project stakeholder shall be able to pick the analyses and compose them as required to perform his investigation. Stakeholders range from software and design engineers to software test engineers, to quality assurance or project leaders.

In [12], we sketched the basic idea of *software analysis as a service*: getting easy access to different analyses from various tools and providers using web services. In the mean time we have experimented with a few implementations of this idea and now can present what we consider a feasible architecture for distributed analysis services. Therefore, the contribution of this very paper is the detailed presentation of the architecture for Software Analysis as a Service, its design considerations and implementation aspects, as well as the set of actually implemented and ready-to-use services based on concrete usage scenarios.

SOFAS follows the principles of a RESTful architecture [7] and allows for a simple yet effective provisioning and use of software analyses based upon the principles of Representational State Transfer around resources on the web. Our architecture is made up by three main constituents: Software Analysis Web Services (*SA-WS*), a Software Analysis Broker (*SA-B*), and Software Analysis Ontologies (*SA-Ontos*). *SA-WS* "wrap" already existing analysis tools as standard RESTful web service interfaces. The *SA-B* acts as the services manager and the interface between the services and the users. It contains a catalogue of all the registered analysis services with respect to a specific software analysis taxonomy. *SA-Ontos* define and represent the data consumed and produced by the different services.

The analyses are accessible via a single entry point and easily invokable by information such as the URLs of the

source control repository, the issue tracking system, or release notes, etc. The user can then compile a "workflow" of the analyses that are required for a particular task. The *SOFAS* platform will take care of actually calling the different services and returning the final results.

Next, we will describe the constituents of the *SOFAS* architecture, its main components and their interaction. Then we will explain how we represent and structure the data produced by different analyses and domains in a homogeneous way. Finally, we will show by means of a working example how *SOFAS* actually works.

II. The *SOFAS* Architecture

In the past years, our group has devised many studies on software evolution and software analysis. The tools we developed and the knowledge we gained are the backbone of a software evolution analysis platform called Evolizer [10]. However, while implementing it and struggling to integrate data produced by other tools for specific analyses, we realized that a big potential lies in having analyses easily accessible and composable, without platform and language limitations, and not having to install and configure particular tools.

SOFAS follows the principles of a RESTful architecture (as introduced by Fielding [7]) and allows for a simple yet effective provisioning and use of analyses based upon the principles of Representational State Transfer around resources on the web. Software analyses are no longer bound to integrated development environments such as Eclipse or other IDEs, but they are accessible on the web on a common web architecture, shown in Figure 1. This architecture is made up by three main constituents: Software Analysis Web Services (*SA-WS*), a Software Analysis Broker (*SA-B*), and Software Analysis Ontologies (*SA-Ontos*). *SA-WS* "wrap" already existing analysis tools by exposing their functionalities and data through standard RESTful web service interfaces. The *SA-B* acts as the services manager and the interface between the services and the users. It contains a catalog of all the registered analysis services with respect to a specific software analysis taxonomy. As such, the domain of analysis services is described in a semantical way enabling users to browse and search for their analysis service of interest. *SA-Ontos* define and represent the data consumed and produced by the different services. Upper ontologies represent generic concepts common to several specific ontologies, providing semantic links between them. In the following we describe each of these three components.

A. Software Analysis Web Services

We use web services over other competing middleware technologies as it is a standard and offers many of the features we need: language, platform and location independence and ease of use. Moreover, we use a RESTful architecture since its very core properties are highly beneficial for our purposes, as we will explain.

1) Architectural considerations for SOFAS: Early prototypes of *SOFAS* were based on classic SOAP RPC-based web services. However, while they can be powerful, the rationale behind them is still highly "application dependent." They were basically created to provide web-based, language independent versions of standard applications, through the use of remote procedure calls (or remote invocations). This means that services may expose any set of operations defined with an arbitrary vocabulary of nouns and verbs, just as applications (e.g. `getUsers()`, `getAnalysis(String analysisName)`). Moreover, since HTTP is used only as a means of transportation, many useful HTTP capabilities, i.e. authentication, content type negotiation, caching, etc., are ignored only to then be re-invented by the service designer as specific methods, overloading the service with yet more arbitrary and heterogeneous methods. Examples are the addition of methods such as `getUsers(String usersListFormat)`, `getUsersAsXML()` or `getAnalysis(String analysisName, String userName, String userPassword,...)`.

The goal of our approach is to provide software analyses—and in particular the data produced—in a simple, generic standardized way, hiding all the peculiarities of the tools actually implementing them. The software analyses we address, typically are linear in the way they work and, more importantly, they behave almost exactly the same way: they need some information about the software project and then run the analysis (be it the code, its source code repository, etc.); once the analysis is done the data produced can be fetched in different, specific formats and the analysis data itself can be updated or deleted. The use of SOAP RPC-based web services would have thus been, in our case, a counterproductive solution, adding unnecessary complexity. The main requirements and characteristics of our services were indeed some of the main inherent principles of REST.

RESTful web services are based directly on HTTP without any additional layer or protocol. They can thus maximize the direct use of the pre-existing, well-defined interface and other built-in capabilities provided by HTTP, minimizing the addition of new application-specific features on top of it. Therefore, in contrast to classic SOAP RPC-based web services, they can use the existing, known standard rich vocabulary of HTTP methods, Internet media types, URIs and response codes. Moreover, they can also directly exploit HTTP caching, user authentication, content type negotiation, etc. To put it in a nutshell, a RESTful web service provides a uniform interface to the clients, no matter what it actually does. It is a collection of resources all identified by URIs, which can be accessed and manipulated with HTTP methods (e.g., POST, GET, PUT or DELETE). Moreover, every message exchanged is self-descriptive as it always contains

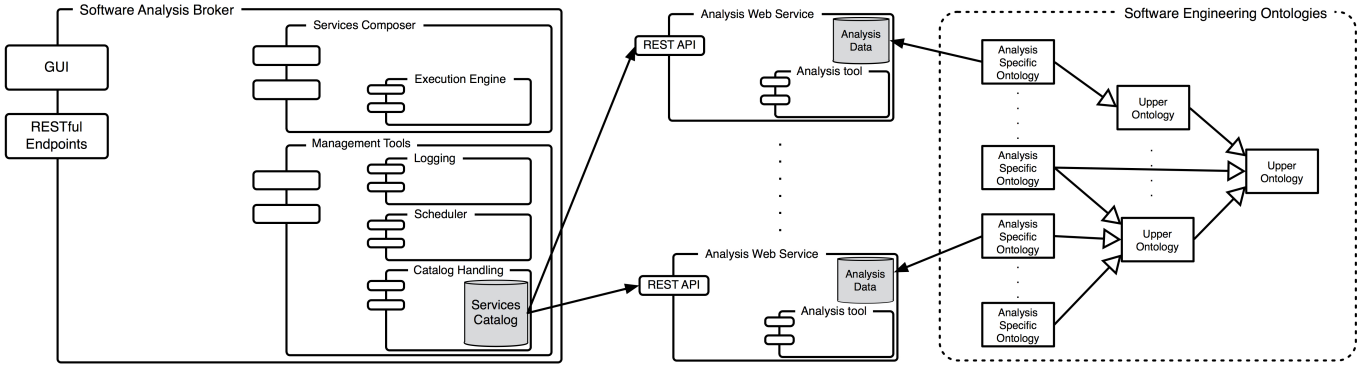

Figure 1. *SOFAS* overall architecture

the Internet media type of the content, which is enough to describe how to process it.

The example SOAP-RPC methods we showed before, in the case of RESTful services would boil down to only one HTTP method, a GET on either the URI identifying the users (e.g. GET http://svexample.com/users) or the analysis (e.g. GET http://svexample.com/analysis_1). The HTTP content type negotiation and access authentication will take care of limiting access to the right users and returning the data in the required format. The combination of specific software analysis services and REST allows us to provide a truly uniform, standard and straight forward interface to those services.

2) The SOFAS Implementation: All services expose two types of resources: the service itself (*e.g.,* http://seal.ifi.uzh.ch/svnImporter/analyses) and the individual analyses (*e.g.,* http://seal.ifi.uzh.ch/svnImporter/analyses/analysis_1). The following methods are available on the service URI:

GET*:* Lists all the existing analyses either in a simple XML-based list or an HTML table, depending on the requested Internet media type.

POST*:* Creates and runs a new analysis. The new analysis URI is assigned automatically and returned by the operation.

On any specific analysis URI, the following methods are available:

GET*:* This method behaves in two ways. If the analysis URI contains a query string, *e.g.,* http://seal.ifi.uzh.ch/svnImporter/analyses/analysis_1?query=‘‘⟨actual_query⟩”, that string will be interpreted as a SPARQL [31] query to fetch specific data from the analysis. The result will be returned in the standard SPARQL Query Results XML Format [2]. This functionality is also known as *SPARQL Endpoint*. In case no query is encoded in the URI, the method just retrieves a representation of the entire addressed analysis, expressed in RDF [21]. We use RDF and its associated query language SPARQL, because we describe all the data produced by the analyses with ontologies. We will explain ontologies in Section II-C.

HEAD*:* Checks if the addressed analysis exists, and if so, if its data is already available. In fact, analyses can take a considerable amount of time, and thus their data might only be available upon their completion. If the analysis does not exist a NOT_FOUND (404) status code is returned, if it exists but it has not completed yet FORBIDDEN (403) is returned. OK (200) is returned if it exists and it has completed successfully.

PUT*:* Replaces the addressed analysis, or if it does not exist, creates and runs it.

DELETE*:* Deletes the addressed analysis.

B. The existing SOFAS services

So far, *SOFAS* contains all the analysis services shown in the architecture overview in Figure 1 and a few more. They are as follows:

Version history services for CVS, SVN, and GIT*:* They extract the version control information comprising release, revision, and commit information from CVS, SVN and GIT repositories: who changed when/which source file and how many lines have been inserted/deleted. In order to work, these services only need the URL of the repository and valid user credentials (username and password). Additional options to further fine-tune the data extraction are also available. For example, extracting the history of just a specific revision interval or fetching the content of every file revision of specific file types. The latter option is particularly useful when additional analyses need to be performed on the actual source code (*e.g.,* model extraction, metrics calculation, etc.).

Meta-model extraction service*:* Given just the source code of a software system, it extracts its static structure in the form of a FAMIX model [33] (a language independent meta-model describing the static structure of object-oriented software). The service is able to partially reconstruct the static structure even when the source code does not compile or has errors, by applying the heuristics already developed for ZBinder [30].

Version history meta-model service: Given a version history extracted by any version history service, it extracts the FAMIX model of all the existing or of a selected set of releases. The model reconstruction works exactly as the previous service.

Metrics service: It computes the most common software metrics from a software system. This service accepts two types of inputs: raw source code or FAMIX meta-models created by the aforementioned FAMIX services. In the current version, it computes these metrics:

- Fan-In and Fan-Out of classes, methods and packages.
- McCabe's cyclomatic complexity [23] of classes, methods and packages.
- Lines of code (LOC) of classes, methods and packages.
- Number of calls in the entire system.
- Height of inheritance tree of classes (HIT).
- Average hierarchy height of the entire system (AHH).
- Average number of derived classes of the entire system (ANDC).
- Number of direct sub-classes of a classes (NDC).
- Number of methods overriding a method in any one of the super-classes of a class (NORM).
- Number of classes (NOC).
- Number of packages (NOP).
- Number of attributes (static and non) of classes and packages (NOA).
- Number of methods (static and non) of classes and packages (NOM).
- Number of parameters of a method (NOPAR).

If no other piece of information is given to the service, it will compute all the metrics for all the source code entities found. Otherwise, the user can set the service to compute only specific metrics for selected entities.

Change Coupling service: Given the version history of a software project, it extracts the change couplings for all the files as described by Gall *et al.* [11]. This means that for every versioned file, it extracts what other files were simultaneously changed with them, how many times and when. The more two files have changed together, compared to the total number of changes they were involved, the more they are coupled.

Change type distilling service: Given a project version history (extracted by one of the aforementioned services), it extracts, for each revision, all the fine-grained source code changes of each source code file. These changes are then classified following the change types taxonomy proposed in [9]. The algorithms used to extract these changes are also based on the ones developed by Fluri *et al.* in the aforementioned paper for the original Change Distiller tool [10].

Issue tracking history services for Bugzilla, Google Code, Trac, and SourceForge: They extract all the historical issue tracking information (problem reports and change requests) from a given issue tracking repository. This data is usually used as-is or together with the project version control information. In the first case, it can help assess the average bug-fixing time, the distribution of bug severity, etc. In the second case, it can be used for more complex analyses, such as location of fault prone files, location and analysis of bug fixing changes, bug prediction, etc. As for the version control services, also this one can be set to import just a range of issues, instead of the whole history.

Issue-revision linker services: Given the issue tracking and version histories of a specific software project, they reconstruct the links between issues and the revisions (also known as commits) that fixed them. As of now three of these services exist for the different heuristics proposed by Mockus *et al.* [25], Sliwerski *et al.* [32], and Fischer *et al.* [8]. Note that all these services structure the extracted data following specific ontologies, which we explain next.

C. Software Analysis Ontologies

We described how REST provides us a truly uniform interface to describe all the analysis services in our architecture, the structure of their input and output and how to invoke them at a syntactic level. However, there is no way to programmatically know what a service actually offers and what the data it consumes/produces means. We address this problem by exploiting semantic web technologies, in particular OWL. An ontology is a formal description of the important concepts (classes of objects) identified in the domain of discourse and their relationship to one another [13]. It provides a common vocabulary for a specific domain, which can be used to express the meta-data needed to capture the knowledge of the exchanged, shared or reused data. Ontologies help tackling both problems, *i.e.,* meaningful service descriptions and data representation.

With OWL we can assign input and output data a clear semantics and a precise syntax, as it is a standardized XML based language. It offers some highly beneficial ontological properties: (1) heterogenous domain ontologies can be semantically "linked" to each other by means of one or more upper ontology, describing general concepts across a wide range of domains. In this way it is possible to reach interoperability between a large number of ontologies accessible "under" some upper ontology. In terms of software analysis services, it means that results from disparate types can be automatically combined given that they share some common concepts; (2) the OWL Description Logic foundation enables automatic reasoning and derive additional knowledge; (3) we can use a powerful query language such as SPARQL; and (4) in contrast to XML and XQuery, which operate on the structure of the data, OWL treats data based on its semantics. This allows for an extension of the data model with no backwards compatibility problems with existing tools.

To describe the data produced by software analyses we developed our own family of Software Evolution ONtologies (*SEON*). The goal is to describe in a clear and univoque

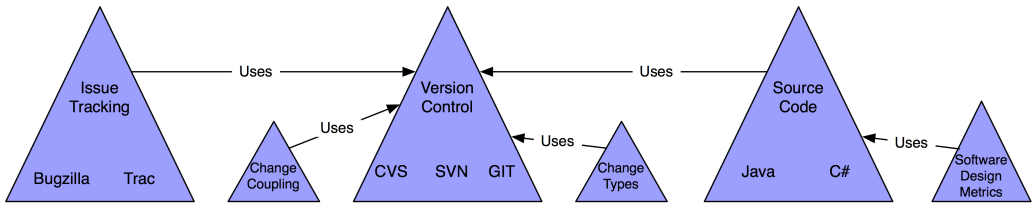

Figure 2. *SEON* overall structure.

way different aspects of software and its evolution, such as version control, issue tracking, static source code structure, change coupling, software design metrics, etc. Figure 2 depicts the basic structure of *SEON*. As of now, the domains described are only the ones addressed by the existing analysis services. For each of the three major subdomains (represented as individual ontology pyramids) we have developed higher level ontologies defining their common concepts. For system-specific or language-dependent concepts we developed some concrete low-level ontologies. But the different ontologies share some concepts and properties. More specifically, the source code, issue tracking, change types and change coupling ontologies use concepts of the version control system one, as the metrics ontology does from the source code one. The version control pyramid can be thus considered the core of *SEON* as it interconnects the three major subdomains.

The *issue tracking ontologies* (for CVS, SVN and GIT) add only few additional concepts to the generic ontology. The SVN ontology, for example, adds the concepts of copies, moves and renames as these operations are poorly supported (or not at all) by others systems. The system specific ontologies introduce additional concepts as the two systems have a different way of classifying bug and issue priorities. Moreover some systems might have a slightly richer or different issue description. For example, Bugzilla keeps track of OS and hardware under which the issue was experienced while Trac does not.

The *source code ontology* models all the static source code structures based on the FAMIX meta model. We decided to use FAMIX instead of other meta models such as UML, as it has a finer granularity and offers more details. As FAMIX was already devised as a language independent source code model for OO programming languages, we represent all the important concepts in the generic ontology. We created the Java and C# ontologies just to address the few particularities. The central concept of this ontology is the class. Through a class the source code ontology can be linked to a version control history, as classes are contained in versioned files.

The *metrics, change types and change coupling ontologies* are simple as they describe rather basic and unstructured data. The first one classifies common software product metrics such as [4], [23]. All these metrics are either computed at package, class or method level. The concept of a metric itself is associated to the concept of an entity of the source code ontology, which represents both classes and methods. The change types ontology describes the source code change types according to the taxonomy proposed by Fluri *et al.* [9]. The change coupling ontology describes how intense two files are coupled in a project: how many times they were changed together during a specific release period.

SEON is continuously evolving. We envision many other ontologies to be incrementally added as new services are provided, for example, ontologies targeting other code quality measurements such as code clones or code smells. Furthermore, ontologies might describe different version control systems (*e.g.,* Mercurial), issue tracking systems (*e.g.,* Jira, Mantis) and programming languages (*e.g.,* C++, Eiffel, Python). The expansion and referencing of existing ontologies or the creation of new ones can be done without changing the already existing ontologies. This is due to the nature of semantic web ontologies: a continuously growing distributed network of loosely interlinked, expandable ontologies. Figure 3 sketches three of the ontologies we just introduced.

D. Software Analysis Broker

Web services enable the sharing, using, and combining different analyses through the net. But they need to be kept track of, classified in a registry, queried, monitored and coordinated. The Software Analysis Broker (*SA-B*) takes care of that, so that the user does not have to interact directly with the raw services. As shown in Figure 1, the *SA-B* is made up of four main components: the *Services Catalog*, a series of management tools, the *Services Composer*, and a user interface.

1) User Interface: The UI is the actual access point to the *SA-B*. It consists of a web GUI, meant for human users and a series of RESTful service endpoints to be (semi)-automatically used by applications. Through the UI the user can easily browse through the *Services Catalog* to check for analyses offered and to select some of them. Apart from the catalog, the user can also pick from some already predefined combinations of analysis services provided as high level analyses workflows (called *analysis blueprints*). Once the desired services are selected, the user might need to set some service-specific settings. Moreover, if the user chose to combine two or more services into a workflow,

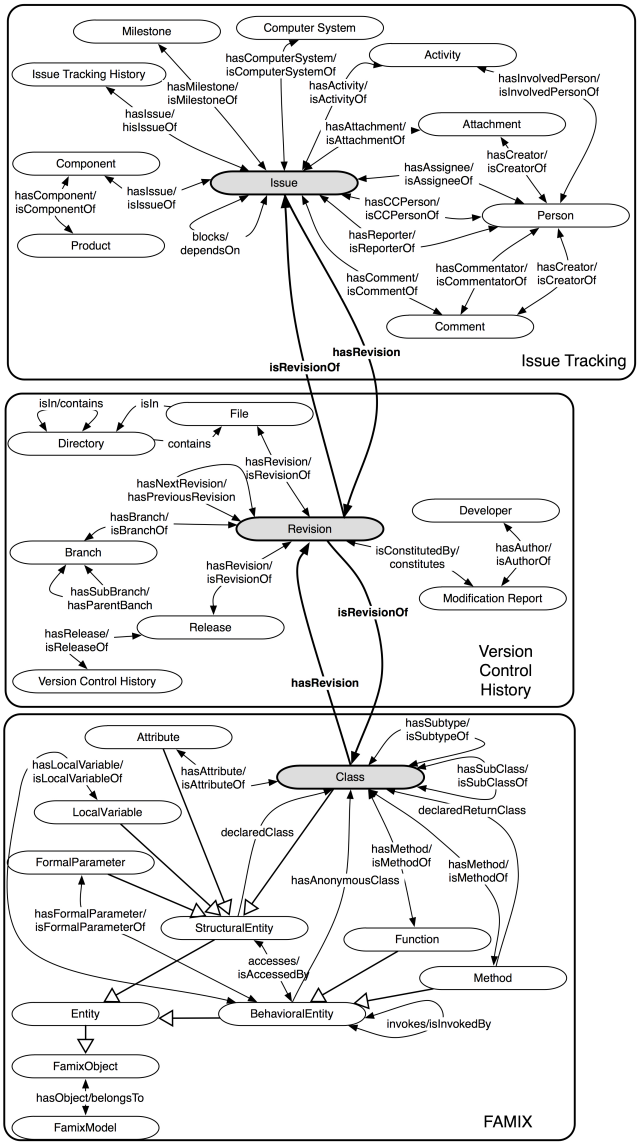

Figure 3. Overview of three of the major *SEON* ontologies.

she would need to actually define how to do that. That is, what their sequence is, what output of a service should be fed as input to another service, etc. The user interface offers an intuitive, high level way to do that, allowing the user to combine the services in a "pipe and filter" fashion. The real composition of those services into an executable workflow and its execution is then taken care of by the *Services Composer*.

2) Services Catalog: The *Services Catalog* stores and classifies all the registered analysis services so that a user can automatically discover services, invoke them, and fetch the results. To do that, an unambiguous classification is essential. We developed such a specific software analysis taxonomy to systematically classify existing and future services. This taxonomy divides the possible analyses into three main categories: development process, underlying models, and source code.

Software development analyses are subdivided into those targeting the development history (extraction, prediction and analysis of source code changes and bugs), its underlying process, and the teams involved in it (their dynamics and metrics). *Model analyses* include those targeting the extraction, either dynamic or static, of specific behavioral and structural model representations (UML, FAMIX, call graphs, etc.) and those computing differences between two models. *Code analyses* are further divided into categories such as checking code well-formedness, correctness and quality. For example, the code quality category is then split into subcategories dealing with code security, conciseness, performance, and design. The latter contains, among others, extractors and analyzers of design metrics and code-smells. A full description is beyond the scope of this paper, but for more details we refer to the *SOFAS* website[1].

Since the literature lacks a preexisting taxonomy of this kind, we structured it mainly using the currently existing approaches as a blueprint and so that they would "fit" reasonably well. This means that our *Services Catalog* is one possibility and by no means complete, as in any classification there are always individuals that do not clearly fit in any category or fit in more than one. However, the proposed categories are reasonable enough, in particular from the perspective of a user who wants to find some particular analyses without struggling with many and sometimes obscure categorizations. Our taxonomy is defined as an OWL ontology. Thus the catalog itself ends up being an instance of that ontology and every registered service an instance of a specific class of that ontology. The ontology is managed and stored in a triple-store and accessed using JENA[2], an open source framework meant exactly to allow for the querying, storing and analysis of RDF/OWL data through a high-level, intuitive API.

We decided to develop this lightweight semantic web-based custom solution, instead of using UDDI, the standard solution for web service registries, for several reasons. The most prominent are related on how it deals with taxonomies, how they are defined, how they are used to classify and then fetch services. UDDI's taxonomies are usually rather simple, flat and with a convoluted definition, especially compared to the cleanness and richness one can reach by using OWL. This highly affects the quality and broadness of classifying and subsequently querying services. On the other hand, with OWL the classification can be as complex and specific as we want the taxonomy to be. Powerful query languages such as SPARQL can be used to query the catalog and fetch specific services. With these languages, the querying options

[1]https://seal.ifi.uzh.ch/sofas

[2]http://jena.sourceforge.net/

become manifold: services can then be queried based on what categories they belong to, on any of their attributes, on the attributes of any of the categories they belong to, etc.

3) Services management tools: Typically just calling services or combining them is not enough. In particular, this holds for long running, asynchronous web services. They need, for example, to be logged and monitored to check if they are up and running, if they are in an erroneous state and why, if they have completed a required operation, etc. Even though these functionalities are vital for end users, their use should be as transparent, standardized and automated as possible. Thus, we implemented a series of services that take care of implementing that as services. As a result, calls to them can be easily weaved into a user defined workflow. The *Services Composer* takes care of doing that.

4) Services composer: This component works both as an interpreter and as the engine running the services workflows. It translates the high level service composition workflow defined by a user through the UI into executable processes and runs them. The decoupling between the user composition definition and the actual composition language is useful for two reasons. First, it allows the user to compose services in a intuitive way, hiding the complexity and technicalities of the actual composition and orchestration. Second, calls to additional services can be automatically weaved into a user defined workflow. In our case, the *Services Composer* adds calls to the management services we just introduced.

We decided to rely on our own custom service composition language and execution engine instead of using existing standards—such as WS-BPEL [18] and one of its related engines—for several reasons. The most prominent is that these languages are meant to be used for SOAP RPC-based services, defined using WSDL. A standard description language (called Web Application Description Language, WADL [14]) has only been recently proposed. Most RESTful services still rely on just human-oriented documentation. Thus, it comes as no surprise that no standard composition language exists yet. Custom solutions such as extending BPEL to account for REST [27], describing RESTful services with WSDL 2.0 or creating new ad-hoc languages and tools [28] have been recently proposed. However, they have not really gained ground or have been used outside theoretical case studies.

As shown in Section II-A2, the services in our architecture not only have the same interface, but they also exhibit the same behavior. Analyses can be started, managed and the outcome data be fetched always in the same manner. This allows us to make several assumptions and simplifications in modeling how analyses work and how they can be composed. A full-blown approach based on BPEL or on a BPEL-like solution would thus be counter productive, adding unnecessary complexity. In particular, an analysis services workflow always consists of starting one or more analyses (an HTTP post method on the service URL), waiting for them to finish (Repeatedly calling an HTTP head method on the analysis URL) and, when done, passing the URI of the results to waiting analyses (along with analysis specific options) and so on, until the workflow is completed, as shown in Figure 4. Consequently, our composer only needs to be able to fetch the services the user selects from the catalog, create the actual service calls, interweave between the simple control structures (*i.e.,* loops to wait for an analysis to complete, loops to restart erroneous analyses or error handling procedures) and pass the data produced by a service to any waiting service in a classic "pipe and filer" fashion.

Our solution is based on the use of WADL to describe the analysis services. By reading a service WADL, the composer knows the input data needed and can thus ask the user to provide it. WADL allows also to incorporate textual descriptions of a service, of its methods and their parameters. This is especially useful for human users. Moreover, we slightly expanded the WADL description so that input and output, when needed, can be declared as being described by specific ontologies, in our case the *SA-Ontos* we previously introduced. This is inspired by SAWSDL (Semantic Annotations for WSDL) [6]. Not only this is useful to guide the user in the composition, but the composer itself can thus, once an analysis is picked, suggests additional services to add to the workflow. In fact it can browse the catalog to fetch all the analyses that produce or require that data. For example, if the service chosen required version control history data as input, all the version control history services would be suggested.

The workflows, once created, will then be stored so that they can be rerun and/or modified in the future. These workflow themselves are RESTful services, adhering to the common behavior we outlined earlier in Section II-A. That means that they can then be interacted with as any other analysis service in *SOFAS*. The only difference is that they require as input all the data needed to invoke correctly all the services in the workflow and producing as output the data generate by the services closing the workflow. Also a WADL description will be created for them. In this way, they can also be composed with other services, into even more complex and structured workflows. In the following, we show a first validation consisting of a usage scenario and actual systems that have been analyzed with *SOFAS* services.

III. Validation

Let us show how *SOFAS* can support a user in a concrete software quality analysis task: finding the code smells of the major releases of ArgoUML[3]. Code smells allow one to spot abnormal and suspicious code entities but also to get an overall impression of the system, as shown, for example,

[3]http://argouml.tigris.org/

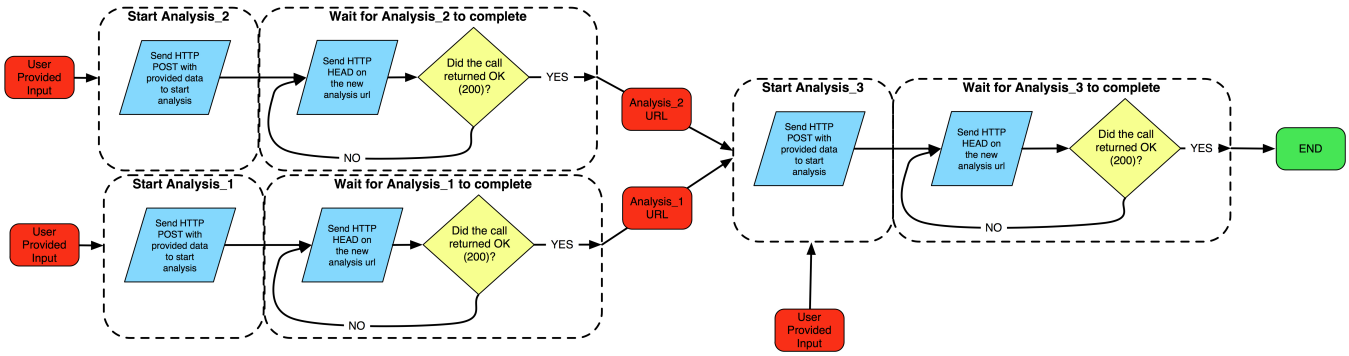

Figure 4. An example of a software analysis workflow.

by Lanza and Marinescu [22]. Moreover, tracking them over a project history helps in assessing the overall quality evolution.

The data to start any analysis involving a system's source code and its history lies in its version control repository (SVN in our case). The first step is thus the invocation of the *SVN version control history service* to extract the full history of the project (since early 1998) along with the source code for all the releases it finds. Once completed, the link to the analysis then is passed to the *Version history meta-model service*. Based solely on that, the service, knowing it is a link to a version control history, is able to automatically fetch the list of releases found and, for each of them, get their source code and reconstruct their FAMIX meta-model (their static structure). The links to each of these models are then passed to the *Metrics service* that, based on each of them, computes the metrics we introduced in Section II-B. These metrics can then be combined to detect smells such as God Class, Feature Envy, etc. As of now, the user has to manually do this last step. However, it is rather a straight forward task, as it is just a matter of combining some of the provided metrics. Nevertheless, an additional service that does that automatically is currently being developed. This service will return, for each code smell the list of code entities (classes and methods) affected. Note that data produced can then also be re-used and fed into other additional services. In our example, the extracted version control history could then be passed to the *Change Coupling service* to find out which classes and files are evolutionary coupled and thus point to other possible architectural weaknesses [5], [11].

SOFAS has already been used internally in our research group for several studies. One Microsoft Surface application uses the data produced by the *meta-model service* for purposes of multi-touch enabled code navigation and design recovery. Another application uses exactly the workflow introduced to visualize multiple evolution metrics as proposed by Pinzger *et al.* [29] on a multitouch screen. For these tools and their evaluation, some of the most popular Java-based open source projects have been analyzed (*e.g.,* ArgoUML, Eclipse, Vuze, jUnit, Tomcat, Derby). Some of the services in *SOFAS* have also been used extensively by external research groups. In particular, all the *version control history services* were used to extract the histories of around 100 open source projects. These projects were a mix of the most known and successful ones (*i.e.,* Python, Gimp, Ruby, or OpenOffice) and the most popular projects in the major OSS forges (*i.e.,* Github, Sourceforge and Tigris). This huge amount of data has been used, for example, to study factors of success and failure in open source projects. More recently it has been used as a base to study contribution and collaboration patterns in OSS projects.

Due to space limitations and to the fact that some of those studies are still yet to be published, the list of all the projects analyzed and details of these studies cannot be fully disclosed at this point.

This is by no means a complete validation of *SOFAS*. However, we claim that it is a serious first proof of the usefulness of the proposed architecture. Furthermore, its already varied and heterogenous usage is a testimony to its versatility, not only for software engineering related tasks. As a matter of fact some of the current users come from very different backgrounds, such as Physics, Management and Economics. More in-depth validations with complex workflows and different usage scenarios will follow, as well as experiments on the properties of services in terms of run-time aspects, data volumes, and reliability.

IV. Related Work

There is a plethora of research works exploiting software project data for software evolution. Approaches focusing on the software evolution either study its source code change history [34], [26], bug history [20], its underlying dynamics [1], [24] or a combination of them [3], [10]. However, all these approaches rely on their own ad-hoc developed tools and techniques and none targeted the issue of using and composing different, independent analyses. Moreover, none of them address the issue of facilitating the analysis usage by thirds by means of web services or similar technologies.

Jin and Cordy [17] were so far the only researchers to study a solution to these issues. They propose an ontology based software analysis tool integration system that employs a domain ontology and specifically constructed external tool adapters. They use a service-sharing methodology that employs a common domain ontology defining the conceptual space shared by the different tools and specially constructed external tool adapters, that wrap the tools into services. They also implemented a proof of concept with three reverse engineering tools that allowed them to explore service-sharing as a viable means for facilitating interoperability among tools. We share with them the overall concept, but at the same time, the two approaches have many differences due to their partially distinct goals. In fact, the objective of their integration effort was to be able to apply a functionality/analysis available in one tool to the fact-base of another one in a very simple way. For this reason, they used a domain ontology just to describe the set of representational concepts that the different tools to be integrated require and support. On the other hand, our goal is to offer a much broader and versatile solution. In fact, we intend to exploit ontologies on a much broader scale: to catalog and describe the services, to represent and standardize their input and output accordingly to the type of analysis offered, to semantically link different results and to perform (semi)-automatic reasoning on them. Moreover their paper just sketches the overall rationale of the approach without going into details on how the proposed architecture was actually implemented and which technologies were used.

The use of web services and semantic web technologies for software analysis, and software engineering in general, has only just recently been addressed in research by just a few works. These works all have focused on providing ontologies to representing software analysis data and concepts to foster software reuse and maintenance. For example, generic software engineering concepts (classes, tests, metrics, requirements, etc.) [16], higher level meta-data about software components (e.g. the programming language, licensing models, ownership and authorship data) [15]. More related to our approach, Kiefer *et al.* [19], developed a software repository data ontology including software, release and bug related information based on based on Evolizer's [10] data models. However none of these models, are then used for concrete software engineering tasks other than a small proof of concept.

V. Conclusions

In this paper we presented *SOFAS*, a flexible and lightweight architecture—both in terms of resources and knowledge requirements—to enable the use and combination of software analyses across platform, geographical and organizational boundaries. We devised these analyses as RESTful webservices accessible through a software analysis broker where users can register, share and use their tools. To enable (semi)-automatic use and composition, these services are classified and mapped into a software analysis taxonomy and adhere to specific meta-models and ontologies for their category of analysis.

We claim that an architecture like the one we devised is highly beneficial for the field of software (evolution) analysis. With very few actions, simple, common analyses can be combined into new, complex and structured ones and then ran. In this way, different stakeholders—which we introduced in the introduction to this paper—could easily extract different type of interesting and useful data about a software project. A project leader might be able to check the status and health of the project by checking, for example, the amount of bugs per file, their distribution and their lifetime (how long it takes to fix them) to maybe allocate resources where needed. A software quality assurance engineer might use it to check the quality of the code (*e.g.,* with OO metric and clone detectors) and be sure that no "software rotting" is going on, or fix it before it goes out of control. A software engineer might use it on a re-engineering task to extract change coupling between source code files (and their evolution) to detect possible cross cutting concerns, hidden or forgotten business rules, clones or in general classes to be improve code cohesion. He might also use it to extract source code metrics to detect potentially problematic classes or disharmonies (*e.g.,* God Class, Brain Class, Intensive Coupling) to drive future refactoring.

SOFAS is still a work in progress and in continuous evolution. In particular, the service composition is still in an early phase. Service composition into workflows is for now only possible through the *SOFAS* web UI provided, and thus only available for human users. In the future we plan to develop a simple ad-hoc composition language so that workflows can be programmatically sent for execution to *SOFAS*. Moreover, new services will be constantly added as soon as they will be developed. Our research group is the main driving force behind it and the provider of the entire support infrastructure and services. However we recently started to look actively for collaborations with other groups to sharing their knowledge and their tools and thus provide new services to the architecture. This is vital in asserting the success and usefulness of our architecture, as one of its main foundations is indeed the sharing of new and diverse analyses by means of services.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 361–370, 2006.

[2] D. Beckett and J. Broekstra. SPARQL query results XML format. W3C Recommendation, 15 January 2008. http://www.w3.org/TR/rdf-sparql-XMLres/.

[3] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In

ESEC/SIGSOFT FSE, pages 177–186, New York, NY, USA, 2005. ACM.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[5] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering*, 99:720–735, 2009.

[6] J. Farrell and H. Lausen. Semantic annotations for WSDL and XML schema. W3C Recommendation, 28 August 2007. http://www.w3.org/TR/sawsdl/.

[7] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[8] M. Fischer, M. Pinzger, and H. C. Gall. Populating a release history database from version control and bug tracking systems. In *In Proceedings of the International Conference on Software Maintenance*, pages 23 – 32, 2003.

[9] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

[10] H. C. Gall, B. Fluri, and M. Pinzger. Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33, January/February 2009.

[11] H. C. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *In Proceedings of the 2003 Sixth International Workshop on Principles Software Evolution*, pages 13 – 23, 2003.

[12] G. Ghezzi and H. C. Gall. Towards Software Analysis as a Service. In *Proceedings of Evol'08, the 4th Intl. ERCIM Workshop on Software Evolution and Evolvability at the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering*, L'Aquila, Italy, September 2008. IEEE Computer Society.

[13] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199 – 220, 1993.

[14] M. J. Hadley. Web application description language (wadl). W3C Member Submission, 31 August 2009. http://www.w3.org/Submission/wadl/.

[15] H. Happel, A. Korthaus, S. Seedorf, and P. Tomczyk. Kontor: An ontology-enabled approach to software reuse. *Proceedings of the 18th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE 2006)*, 2006.

[16] D. Hyland-Wood, D. Carrington, and S. Kaplan. Toward a software maintenance methodology using semantic web techniques. *Proceedings of the 2nd International IEEE Workshop on Software Evolvability at IEEE International Conference on Software Maintenance (ICSM 2006)*, pages 23–30, 2006.

[17] D. Jin and J. R. Cordy. Ontology-based software analysis and reengineering tool integration: the oasis service-sharing methodology. *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 613–616, 2005.

[18] D. Jordan and J. Evdemon. Web services business process execution language version 2.0. OASIS Standard, 11 April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[19] C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with isparql and a software evolution ontology. *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007)*, 2007.

[20] S. Kim, T. Zimmermann, E. W. Jr., and A. Zeller. Predicting faults from cached history. *Proceedings of the 29th international conference on Software Engineering (ICSE 2007)*, pages 489–498, 2007.

[21] G. Klyne and J. J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation, 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[22] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[23] T. J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[24] A. Mockus and J. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 503–512, 2002.

[25] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 8th International Conference on Software Maintenance*, pages 120–130, 2000.

[26] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 284–292, 2005.

[27] C. Pautasso. Bpel for rest. In *7th International Conference on Business Process Management (BPM08)*, 2008.

[28] C. Pautasso. Composing restful services with jopera. In *International Conference on Software Composition*, pages 142–159, Zurich, Switzerland, July 2009. Springer.

[29] M. Pinzger, H. C. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis'2005)*, pages 67–75, 2005.

[30] M. Pinzger, E. Giger, and H. C. Gall. Handling unresolved method bindings in eclipse. Technical report, Department of Informatics, University of Zurich, Switzerland, 2007.

[31] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C Recommendation, 15 January 2008. http://www.w3.org/TR/rdf-sparql-query/.

[32] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR*, 2005.

[33] S. Tichelaar, S. Ducasse, and S. Demeyer. Famix and xmi. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 296, Washington, DC, USA, 2000. IEEE Computer Society.

[34] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version history to guide software changes. *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, 2004.